# Project 1: Threads

## Preliminaries

> Fill in your name and email address.

寿晨宸 2100012945@stu.pku.edu.cn

> If you have any preliminary comments on your submission, notes for the
> TAs, please give them here.
> Please cite any offline or online sources you consulted while
> preparing your submission, other than the Pintos documentation, course
> text, lecture notes, and course staff.

## Alarm Clock

## DATA STRUCTURES

> A1: Copy here the declaration of each new or changed struct or struct member, global or static variable, typedef, or enumeration. Identify the purpose of each in 25 words or less.

-
  ```
  static struct list sleeping_list;
  ```

  Store sleeping threads.

-
  ```
  struct sleeping_thread
  {
    struct thread* sleep_thread;
    struct list_elem elem;
    int64_t wake_time;
  };
  ```

  As element of `sleeping_list`, which contains the pointer of the thread itself, the true node in the list and the remaining sleeping time.

## ALGORITHMS

> A2: Briefly describe what happens in a call to timer_sleep(),
> including the effects of the timer interrupt handler.

`timer_sleep()`

1. Disable the interrupts.
2. Compute the pointer of the current thread and the time thread should be awakened. Put them in a `sleeping_thread` structure.
3. Insert the struct in the `sleeping_list` by the order of `wake_time`.
4. Block the current thread.
5. Reset interrupts level.

`timer_interrupt()`

- Update the statement of `sleeping_list` and unblock the threads which should be awakened.

> A3: What steps are taken to minimize the amount of time spent in
> the timer interrupt handler?

Insert the node in `sleeping_list` by order. Such that when interrupt handler wants to modify it, it need not travel all the list. It could just stop when the current element's `wake_time` less than the current time `ticks`.

## SYNCHRONIZATION

> A4: How are race conditions avoided when multiple threads call
> timer_sleep() simultaneously?

Call the function `intr_disable()` and `intr_set_level()`, which disable interrupts.

> A5: How are race conditions avoided when a timer interrupt occurs
> during a call to timer_sleep()?

List operations happen while interrupt is disabled.

## RATIONALE

> A6: Why did you choose this design? In what ways is it superior to another design you considered?

Designing a list of sleeping threads is audio-visual. The key is how to organize the list. Ordered or disordered?
In my opinion, whenever the time interrupt occurs, the OS would call `timer_interrupt()`, which is much more frequent than `timer_sleep()`. So optimizing `timer_interrupt()` is more effective. I think I could minimize the total time spent by sorting the list before travel it, since the time spent of `timer_sleep()` would increase.
On other side. If the list organized disordered, it would cost much more time to travel all the list when call `timer_interrupt()` to update it.

So I decide to sort `sleeping_list` when insert the thread. And when `timer_interrupt()` update the list, it would spend less time.

# Priority Scheduling

## DATA STRUCTURES

> B1: Copy here the declaration of each new or changed struct or struct member, global or static variable, typedef, or enumeration. Identify the purpose of each in 25 words or less.

- `struct lock`:

  ```
  /* Record the maximum priority of the threads waiting for this lock; */
  int donate_priority;
  /* Used as a node for a list. */
  struct list_elem elem;
  ```

- `struct thread`:

  ```
  /* Save the priority of the thread before donation for recovery. */
  int prev_priority;
  /* The lock current thread waiting for. */
  struct lock *wait_lock;
  /* The list of locks held by the current thread, arranged in descending order b
  struct list hold_locks;
  ```

- `synch.h`:

  ```c
  /* Maximum search depth of Nested priority Donation. */
  #define NEST_PRI_DONATE_DEPTH 8
  ```

> B2: Explain the data structure used to track priority donation.
> Use ASCII art to diagram a nested donation. (Alternately, submit a
> .png file.)

The data structures mentioned in **Q:B1** would all be used to track priority donation.

Behaviors relatred to **DONATION** only occur when a thread acquire and release a lock.

Firstly, when we initial a thread, we set the **wait_lock** to NULL, the **prev_priority** to priority. And when we initial a lock, we set the **donate_priority** to PRI_MIN.

The lock's **elem** member was used in the thread's **hold_locks** list.

As for single donation case, we would check the lock's holder's priority when the lock is being acquired. If current thread's priority is greater than the lock's holder's priority, donation happens.

Thread's **prev_priority** will change with priority except donation, such that the **prev_priority** would record the priority before donation. When donation happens, the donated thread's priority and lock's **priority_lock** (which keep track of the maximum priority of threads waiting for the lock) would be set to donate thread's priority; And the donate thread's **wait_lock** is set to this lock.

Then we would focus on nested donation. Let's check whether the donated thread is blocked by another lock. If so, another donation would happen again by the algorithm explained above. We replace new donate thread with the last donated thread, and replace the new donated thread with the holder of the lock acquired by the last donated thread. The nested donation situation would go on iteratively untill no donated thread is blocked by any lock or it reaches the max depth(**NEST_PRI_DONATE_DEPTH**, determined how deep we could search for).

Then whenever a lock is obtained, the lock would be inserted into the thread's **hold_locks** list in descending order by lock's **donate_priority**. And we should set the lock's **donate_priority** to PRI_MIN.

When we release a lock, eventualy the lock would be removed from the holder's **hold_locks** list. Then we need to restore the priority. If the **hold_locks** list is empty, we just need to restore the priority to **prev_priority**.Else, it means multiple donation happens. We should get the front element of the list(which has the max **donate_priority**),and set the priority to the maximum of front element's **donate_priority** and **prev_priority**.

And we would set the lock's **donate_priority** to PRI_MIN. Don't worry, if some thread get it soon, lock's priority would be set to the thread's priority, which must be the max priority of threads waiting for it. And if the lock truly be free, it should be reset to PRI_MIN too.

Using the data structure and algorithm above, we could implement the sigle donation, multiple donation, and nested donation. As following, I would show a simple example of a nested donation using diagram.

Example:

- **thread L**, priority 31, hold lock_1.
- **thread M**, priority 32, hold lock_2 and acquire lock_1.
- **thread H**, priority 33, acquire lock_2.

---

**Step 1: At the beginning:**

- **thread L**:

| member | value |
|---|---|
| priority | 31 |
| prev_priority | 31 |
| hold_locks | {lock_1(donate_priority=31)} |
| wait_lock | NULL |

- **thread M**:

| member | value |
|---|---|
| priority | 32 |
| prev_priority | 32 |
| hold_locks | {lock_2(donate_priority=32)} |
| wait_lock | NULL |

- **thread H**:

| member | value |
|---|---|
| priority | 33 |
| prev_priority | 33 |
| hold_locks | {} |
| wait_lock | NULL |

---

**Step 2: thread M acquires lock_1:**

- **thread L**:

| member | value |
|---|---|
| priority | 32 |
| prev_priority | 31 |
| hold_locks | {lock_1(donate_priority=32)} |
| wait_lock | NULL |

- **thread M**:

| member | value |
|---|---|
| priority | 32 |

| member | value |
|---|---|
| prev_priority | 32 |
| hold_locks | {lock_2(donate_priority=32)} |
| wait_lock | &lock_1 |

- **thread H**:

| member | value |
|---|---|
| priority | 33 |
| prev_priority | 33 |
| hold_locks | {} |
| wait_lock | NULL |

---

**Step 3: thread H acquires lock_2(In depth 1):**

- **thread L**:

| member | value |
|---|---|
| priority | 32 |
| prev_priority | 31 |
| hold_locks | {lock_1(donate_priority=32)} |
| wait_lock | NULL |

- **thread M**:

| member | value |
|---|---|
| priority | 33 |
| prev_priority | 32 |

| member | value |
|---|---|
| hold_locks | {lock_2(donate_priority=33)} |
| wait_lock | &lock_1 |

- **thread H**:

| member | value |
|---|---|
| priority | 33 |
| prev_priority | 33 |
| hold_locks | {} |
| wait_lock | &lock_2 |

---

**Step 4: thread H acquires lock_2(In depth 2):**

- **thread L**:

| member | value |
|---|---|
| priority | 33 |
| prev_priority | 31 |
| hold_locks | {lock_1(donate_priority=33)} |
| wait_lock | NULL |

- **thread M**:

| member | value |
|---|---|
| priority | 33 |
| prev_priority | 32 |
| hold_locks | {lock_2(donate_priority=33)} |

| member | value |
|---|---|
| wait_lock | &lock_1 |

- **thread H**:

| member | value |
|---|---|
| priority | 33 |
| prev_priority | 33 |
| hold_locks | {} |
| wait_lock | &lock_2 |

---

**Step 5: thread L released lock_1:**

- **thread L**:

| member | value |
|---|---|
| priority | 31 |
| prev_priority | 31 |
| hold_locks | {} |
| wait_lock | NULL |

- **thread M**:

| member | value |
|---|---|
| priority | 33 |
| prev_priority | 32 |
| hold_locks | {lock_2(donate_priority=33), lock_1(donate_priority=32)} |
| wait_lock | NULL |

- **thread H**:

| member | value |
|---|---|
| priority | 33 |
| prev_priority | 33 |
| hold_locks | {} |
| wait_lock | &lock_2 |

---

**Step 6: thread M released lock_2:**

- **thread L**:

| member | value |
|---|---|
| priority | 31 |
| prev_priority | 31 |
| hold_locks | {} |
| wait_lock | NULL |

- **thread M**:

| member | value |
|---|---|
| priority | 32 |
| prev_priority | 32 |
| hold_locks | {lock_1(donate_priority=32)} |
| wait_lock | NULL |

- **thread H**:

| member | value |
|---|---|
| priority | 33 |
| prev_priority | 33 |
| hold_locks | {lock_2(donate_priority=33)} |
| wait_lock | NULL |

## ALGORITHMS

> B3: How do you ensure that the highest priority thread waiting for
> a lock, semaphore, or condition variable wakes up first?

Implemet a sorted list, which is in descending order of priority. Whenever I wake up a thread, I just need to pop the front element of the list.

> B4: Describe the sequence of events when a call to lock_acquire()
> causes a priority donation. How is nested donation handled?

The function called when a thread acquire a lock. The things a thread would do are the following.

1. **Check the lock and donation.**
    1. Check the lock we acquire whether held by other thread.(**lock->holder**?) If so, go on, else just go to the part **GET the lock**.
    2. Set current thread's struct member **wait_lock** as this lock, prepare for the nest donation.
    3. Update the lock's struct member **donate_priority**, which means the max priority the threads waiting for it have. Then, sort the lock's holder's **hold_locks** list, which make sure it is in descending **donate_priority** order.
    4. **Single donation**. If current thread's priority greater than the lock's holder's priority, donate to the holder thread. Set holder thread's priority to current thread's priority. As for the **prev_priority**, don't change it, it would record the thread's priority before donation.
    5. **Nested donation**. If H is waiting on a lock that M holds and M is

    waiting on a lock that L holds, then both M and L should be boosted to H's priority. So we should search the threads chain in depth of **NEST_PRI_DONATE_DEPTH**( I set it to 8). If the threads' priority in the chain is decreasing, we do donate behavior, in which the waiting thread donate to the holder thread, until the priority of the thread rises or we reach the max depth.

2. **Wait for the lock or just decrease the semaphore.**
3. **Get the lock.**
    1. Because the current thread no longer wait for a thread. So we set current thread's **wait_lock** to NULL.
    2. Update the lock's **donate_priority**, set it to current thread's priority.(Because the thread which get lock surely have the max priority in the threads waiting for the lock)
    3. Insert the lock into the thread's hold_locks list in the descending donate_priority order.
    4. Set the lock's holder to current thread.

> B5: Describe the sequence of events when lock_release() is called
> on a lock that a higher-priority thread is waiting for.

1. Remove the lock from holder's **hold_locks** list.
2. Set the lock's **donate_priority** to PRI_MIN.(If the lock be free, we should sure it could be held by any thread).
3. Restore the priority of the current thread. We set it to the maximum of the **prev_priority** and the max **donate_priority** of locks in the hold_locks(just pop the front element due to the list's descending order). Because If the thread hold many locks, we shouldn't only restore the priority to **prev_priority**. Actually we should focus on the situation that multiple threads donate do one thread.
4. Set the holder of the lock to NULL.
5. Increase the semaphore.

## SYNCHRONIZATION

> B6: Describe a potential race in thread_set_priority() and explain
> how your implementation avoids it. Can you use a lock to avoid
> this race?

When donation happens, other threads would change the donated thread's priority. At the mean time, the thread itself would call `thread_set_priority()` to change its priority. Then race occurs.

To prevent the race happens, we disable the interrupt by call `intr_disable()`.

Yes, but I need to add another lock element in the struct thread, which shared by the donate priority and the donated priority. Whenver the thread itself or the donate thread want to change its priority, they should acquire the added lock first. (At any time, there would be only two threads want to acquire the added lock, so we don't need to focus on other complex race condition) Of couse, we need to handle the nested call of locks, so we need to judge if the acquire lock we have already held.

## RATIONALE

> B7: Why did you choose this design? In what ways is it superior to
> another design you considered?

1. Implement all list as ordered list rather than disordered list. To minimize the time cost when search for the max priority element.
2. Use struct member **prev_priority** to record the priority before donation and mark whether the thread have been donated, rather than add another **is_donated** member in the struct thread. Such that we can simplify the structure.
3. Disable the interrupt rather than use another lock to avoid the race, such that our code would be more concise.
4. Handle all the operation of donation when acquire or release a lock, rather than implement other functions. May be the code would be long, but it alse be readable.

# Advanced Scheduler

## DATA STRUCTURES

> C1: Copy here the declaration of each new or changed struct or struct member, global or static variable, typedef, or enumeration. Identify the purpose of each in 25 words or less.

- file `fixed-point.h`

```
/* The type which support floating-point arithmetic */
typedef int fixed_point_t;
/* Use 17.14 representation, this is the size of fractional part */
# define FRACTIONAL_NUM 14

/* And I alse implement other fixed-point arithmetic operations in this file by
```

- `struct thread`

```
/* thread nice value */
int nice;
/* thread recent cpu */
fixed_point_t recent_cpu;
```

- file `thread.h`

```
#define NICE_MAX 20
#define NICE_MIN -20
#define NICE_DEFAULT 0
```

- file `thread.c`

```
/* load_avg for 4.4BSD Schedule */
fixed_point_t load_avg;
```

## ALGORITHMS

> C2: How is the way you divided the cost of scheduling between code inside and outside interrupt context likely to affect performance?

The call of timer interrupt is frequently. And we need to update the **load_avg**, **nice**, **recent_cpu** and **priority** while the timer ticks. So if CPU spends much time to manage interrupt, the normal threads would not acquire enough time to run. So they would occupy more CPU time, which means the higher **load_avg**, **recent_cpu** and lower **priority**. It would probably lower performance.

So I choose to minimize the cost of timer interrupt by using ordered list to handle sleeping threads and avoiding do anything expect which mentioned above.

## RATIONALE

> C3: Briefly critique your design, pointing out advantages and
> disadvantages in your design choices. If you were to have extra
> time to work on this part of the project, how might you choose to
> refine or improve your design?

I only implement one ordered **ready_list** , which meeas we need to sort it whenever any thread's priority has been modified or we insert any new threads. It would spend much time ( O(nlgn) ).

However, if I implement an array contains 64 list which store thread having correspond priority, it would be much more convenient. When we modify any thread's priority, we just need to remove it from one list and push it in another. When we have to create/block/unblock/field a new thread, we just need to choose the correspond list and do it. Only when we choose the next thread to run, we need to travel the array to find the first list hold thread, and then pop the front element. All oprations would be O(1).

So I will implement the 64 list of ready threads if I have enough time. ( but I don't want to modify my current implement, because it has already pass all tests)

> C4: The assignment explains arithmetic for fixed-point math in
> detail, but it leaves it open to you to implement it. Why did you
> decide to implement it the way you did? If you created an
> abstraction layer for fixed-point math, that is, an abstract data
> type and/or a set of functions or macros to manipulate fixed-point
> numbers, why did you do so? If not, why not?

The Pintos dosn't support the float calculations, but the **load_avg** and **recent_cpu** are real numbers. So we need to implement an abstraction of fixed-point math to implement and calulate real numbers.

I use macros and new type fixed-point in file fixed-point.h rather than static functions or inline functions. Because all of the operations are simple, we need not to implement complex logic, which means we could do it just use concise macros. And macros are faster than inline functions.